

Proprietà Ricerca Best First Ricorsiva

Come A^* , la Ricerca Best First Ricorsiva è ottima se la funzione euristica $h(n)$ è **ammissibile**

Inoltre:

- Complessità spaziale: $O(bd)$
- Complessità temporale: difficile da definire (ma esponenziale, nel caso pessimo...)

In ogni caso, il problema di RBFS (ed anche IDA^*) consiste nell'uso di **troppa poca memoria**

Gli algoritmi MA^* (*memory-bounded A^**) e SMA^* (*simplified MA^**) utilizzano esattamente **tutta la memoria disponibile**

Simplified MA*

La Simplified MA* procede come A*, espandendo la foglia migliore fino a quando la memoria è piena

Successivamente, per poter proseguire, deve cancellare dalla memoria un nodo **vecchio** per far posto ad un nodo **nuovo**

In particolare:

scarta sempre il nodo foglia **peggiore** (f-valore più alto, cioè quello in fondo alla coda)

l'f-valore del nodo scartato viene **riportato sul nodo padre**

si espande di nuovo il nodo padre quando **tutti gli altri cammini sono peggiori**

Simplified MA*: complicazione...

Cosa succede se *tutte* le foglie hanno lo stesso f-valore??

SMA* rischierebbe di scegliere lo stesso nodo sia per la cancellazione che per l'espansione!

Politica adottata:

rimuove la foglia **più vecchia** ed espande la foglia **più recente**

se queste **coincidono**, significa che **la soluzione che passa da quel nodo non può essere contenuta dalla memoria disponibile...**

... e quindi **è giusto rimuoverla!**

Quindi SMA* è

- **completa solo se** la soluzione può essere contenuta in memoria
- **ottima** se la soluzione è **raggiungibile**
... altrimenti restituisce la **migliore soluzione raggiungibile**

Algoritmi di miglioramento iterativo

In molti problemi di ottimizzazione, *il cammino* è irrilevante;
la soluzione è costituita dallo stato goal stesso

Quindi lo spazio degli stati è dato dall'insieme delle
configurazioni “complete”;

trovare una configurazione *ottima*, es.: TSP

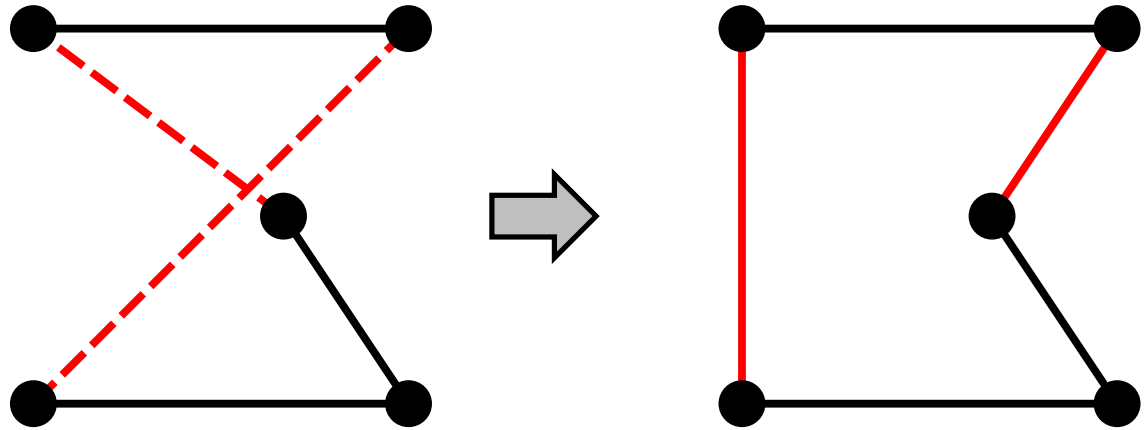
o, trovare una configurazione che soddisfi dei vincoli, es.: orario

In tali casi, si possono usare gli algoritmi di *miglioramento iterativo*;
Mantengono un singolo stato “corrente”, e tentano di migliorarlo

Impiegano spazio costante, e sono quindi adatti sia per ricerca online che
per ricerca offline

Esempio: Problema del commesso viaggiatore

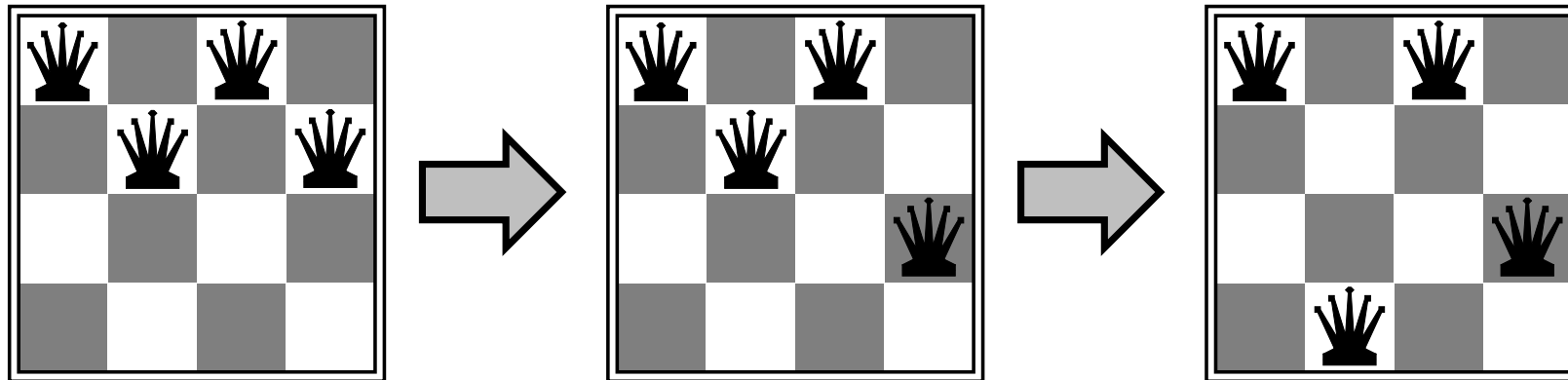
Si parte con un percorso qualsiasi, e si eseguono scambi a coppie



Esempio: n -regine

Disporre n regine su una scacchiera $n \times n$ senza che si minaccino (non ci devono essere due regine sulla stessa riga, colonna, o diagonale)

Muovere una regina in modo da minimizzare il numero di minacce



Hill-climbing (o discesa/ascesa di gradiente)

function HILL-CLIMBING(*problema*) **returns** uno stato che è un massimo locale

inputs: *problema*, un problema

variabili locali: *nodo_corrente*, un nodo
vicino, un nodo

nodo_corrente ← CREA-NODO(STATO-INIZIALE[*problema*])

loop do

vicino ← il successore di *nodo_corrente* di valore più alto

if VALORE[*vicino*] ≤ VALORE[*nodo_corrente*] **then return** STATO[*nodo_corrente*]

nodo_corrente ← *vicino*

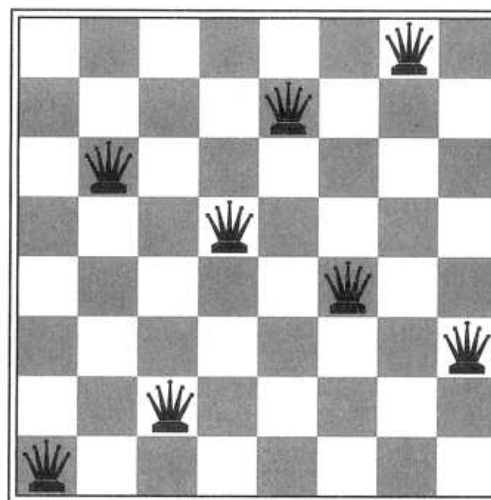
Hill-climbing: esempio delle 8 regine

$h(s)$ = numero di coppie di regine che si attaccano a vicenda

Quanti successori per ogni stato ? 8×7

(# regine \times caselle libere su colonna/riga [deve esserci 1 regina per colonna/riga])

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♚	13	16	13	16
♚	14	17	15	♚	14	16	16
17	♚	16	18	15	♚	15	♚
18	14	♚	15	15	14	♚	16
14	14	13	17	12	14	12	18



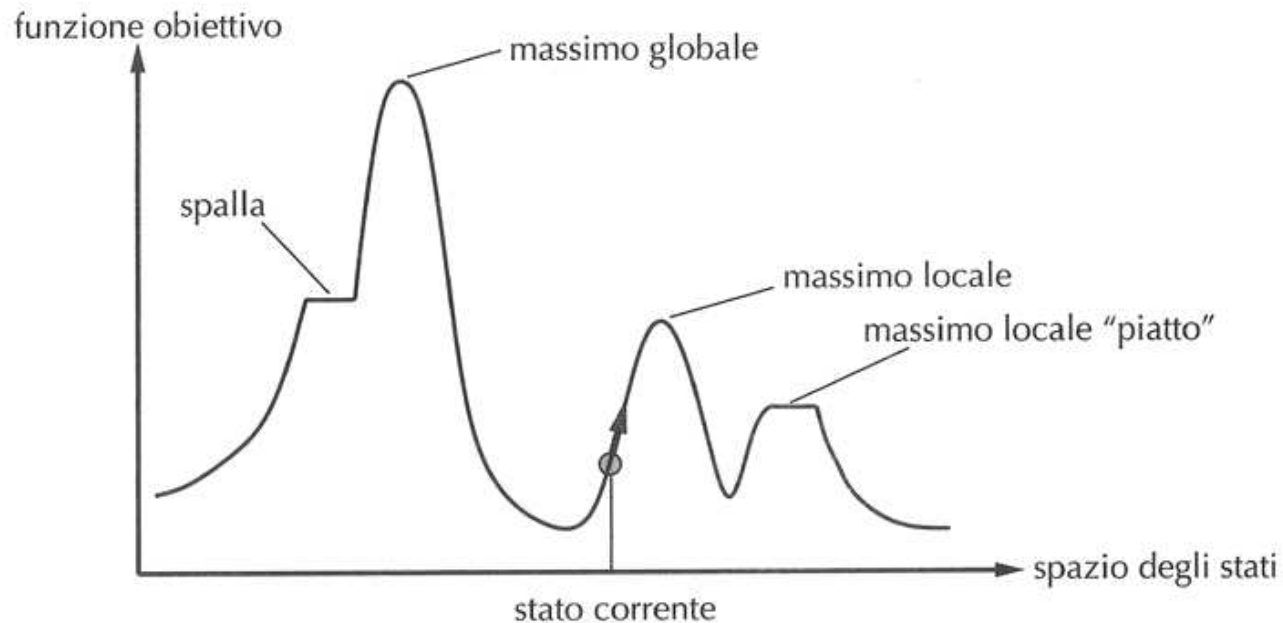
Uno stato con $h = 17$ e valori di h per ogni successore indicati

Un minimo locale con $h = 1$. Ogni successore di questo stato ha $h > 1$.

Lo stato a destra “dista” solo 5 passi da quello a sinistra

Hill-climbing: massimi locali ed altri problemi

Problemi: a seconda dello stato iniziale, può fermarsi su dei massimi locali e/o "spalle"



Nel caso di spazi continui è difficile scegliere la dimensione del passo di progressione ed inoltre si può avere una convergenza molto lenta